Week 10 : Pandas (continued)

pending tasks

type

Dataframe object

- Dataframe objects are a generalisation of series objects. If row and column names are not initialized, they are implicitly indexed as in series.
- Dataframe objects can be created using numpy arrays, multiple series objects.

```
#creating dataframes using numpy arrays
arr = np.random.randint(0, 10, (5, 3))
df = pd.DataFrame(arr)
df.index = ['R1', 'R2', 'R3', 'R4', 'R5']
df.columns = ['C1', 'C2', 'C3']
type(df.iloc[2:4, 1:3])
>>> pandas.core.frame.DataFrame
```

- The names of rows and columns can be obtained using df.index and df.columns and changed by assigning new values. All values can be obtained using df.values.
- loc and iloc can be used for explicit and implicit indexing respectively. Specific cell values or slices of dataframe can be accessed as shown above. In slices obtained using iloc both the end values are included, returning a series. While indexing, first the row is specified, followed by the column.
- Accessing a single row or column returns a series object, thus, dataframes can be thought as a collection of series objects.

Task on creating dataframes

- A generic function can be written to create a dataframe using numpy arrays of given dimensions and range as follows:

```
def create_df(nRows, nCols, maxRand=10):
    arr = np.random.randint(0, maxRand, (nRows, nCols))
    df = pd.DataFrame(arr)
    df.index = ['R' + str(x) for x in np.arange(1, nRows+1)]
    df.columns = ['C' + str(x) for x in np.arange(1, nCols+1)]
    return df
```

- The function DataFrame() is overloaded, thus, depending on the argument type passed different functions are available for dataframe creation.
- Given two series objects: mass and diameter, a dataframe can be created by using a dictionary of column name corresponding series as follows:

```
df = pd.DataFrame({'Mass': mass, 'Diameter': diameter})
```

The resulting dataframe has a union of indices of the series objects. Missing values are assigned NaN values.

- The rows and columns can be accessed using the corresponding indices either by subscripting or using the dot operator. The shortcoming of dot operator being, it cannot be used when the name used is a pandas keyword.

```
# accessing row values
df.loc['Earth']
# accessing column values
df.loc[:,'Mass']
```

Creating mean row

- A new row ‘Col_Mean’ can be created and initialised using loc:

```
df.loc['Col_Mean'] = [np.mean(df['Mass']), np.mean(df['Diameter'])]
```

Here the columns are series objects, thus, numpy functions can be used.

- Alternatively, df.mean() function can be used, which by default, calculates the column-wise mean values. To row-wise computation, the axis had to be specified as df.mean(axis = 1).

```
# example to calculate row and column mean
df['Row_Mean'] = df.mean(axis=1)
df.loc['Col_Mean'] = df.mean()
```

- Other statistical quantities such as df.min(), df.max(), df.median(), df.quantile(0.25) can be used in a similar way. df.describe() does all the above computations for each numerical column and ignores the non-numerical columns.
- Updating the structure of a dataframe using functions such as df.drop() returns a new dataframe. For updating the original df object, the argument **inplace = True** has to be passed.

Working with planetary dataset

- df.info can be used to get an overview of the dataset such as column names, null-counts and respective data types.
- The count returned in df.describe() is count of non-Null values in the respective columns.

Dropping null values

- In the brute force approach, we do cell-wise iteration, iterating through each row, checking if any value is null and dropping if so. This looping can be done as follows:

```
for r in df.index:
    for c in df.columns:
        if pd.isnull(df.loc[r, c]):
            df.drop(r, inplace=True)
            break
```

- Alternatively, row-wise iteration can be done using df.iterrows() which returns the row index along with the row series object as follows:

```
for i, r in df.iterrows():
    if pd.isnull(r).any():
        df.drop(i, inplace=True)
```

any() function does an ‘or’ operation for the given function isnull().

- There exists a function df.dropna() in pandas which drops the rows containing null values.

```
df.dropna(inplace=True)
```

Querying from dataframe

Question:

(using planets dataset from seaborn package)

Filter and show only those rows which have planets that are found in the 2010s and method is 'Radial Velocity' or 'Transit' and distance is large (> 75 percentile).

solution:

- Brute force approach is to iterate and drop the rows that do not satisfy the conditions. The following code snippet demonstrates this approach.

```
for i, r in df_.iterrows():
    if r['year'] < 2010:
```

```
        df_.drop(i, inplace=True)
        continue
    if r['method'] != 'Radial Velocity' and r['method'] != 'Transit':
        df_.drop(i, inplace=True)
        continue
    if r['distance'] < per_75:
        df_.drop(i, inplace=True)
        continue
```

- Queries are written to select subsets of data satisfying given constraints from a database.
- Dataframe objects can be indexed by using conditional checks on each of the column.

```
df_ = df_[
    (df_['year'] >= 2010) &
    ((df_['method'] == 'Radial Velocity') | (df_['method'] == 'Transit')) &
    (df_['distance'] > per_75)
]
```

Applying functions to dataframes

Question:

Modify the method column to have only the abbreviation of each method.

solution:

- df[‘method’].unique() can be used to view the unique values in the column ‘method’.
- An approach can be to create a dictionary of key value pairs of original string and corresponding abbreviation as the value. Then iterate through the rows and map the ‘method’ names to dictionary values.

```
# using iterrows and a dictionary of key value pairs.
short_names = {}
for s in df.method.unique():
    short_names[s] = ''.join([x[0] for x in s.split(' ')])
for i, r in df.iterrows():
    df.loc[i, 'short_method'] = short_names.get(r['method'], r['method'])
```

- An alternate method is to use .apply() that applies a function to the all the column values. This can be used when the same operation is done on all the rows.

```
# using .apply()
def shorten_method(s):
    return short_names.get(s, s)
df['short_method'] = df['method'].apply(shorten_method)
```

Use of groupby method

Template:

1. **Split** the dataframe into smaller chunks (in this case they should have the same method name)
2. **Apply** some function in each smaller chunk (in this case it is the count function)
3. **Aggregate** the results from each chunk together

Question:

Count the number of planets discovered for each method type.

solution:

- The above logic can be implemented as follows:

```
d = {}
for m in df.method.unique():
    d[m] = df[df.method == m]['method'].count()
```

- Pandas provides groupby() method using which the same can be achieved.

```
df.groupby('method')['method'].count()
```

In both the above cases, other statistical functions such as mean() can be applied.

- On a side note, number rows corresponding to each unique value in a column can be calculated using value_count() as follows:

```
df['method'].value_counts()
```

Filter, split, apply, aggregate

Template:

Filter the data for given condition (in this case planet found in last decade)

1. **Split** (in this case across method)
2. **Apply** (in this case just count)
3. **Aggregate** (to represent the final result)

Question:

Find out what fraction of planets have been found in the last decade (i.e., in 2010s) across each method type.

solution:

```
# step1:  filter the required data
df[df.year >= 2010]
# step 2 : groupby() and apply the count function to method column
s_2010s = df[df.year >= 2010].groupby('method')['method'].count()
s_allTime = df.groupby('method')['method'].count()
# aggregate the output in required format
s_2010s/s_allTime
```

Working with Nifty50 dataset

- Dataframes can be read from csv filed using pd.read_csv(file_name')
- pd.concat([list_of_dataframes]) can be used to concatenate dataframes by rows, aligned on the column index. The parameter axis=1 can be set to align on the row index. Hierarchical indexing is supported by setting the **keys** parameter. Accessing such data can be done as follows:

```
# concatenating the additional columns to each row
nifty_2019 = pd.concat([nifty50_2019, niftynext50_2019], axis=1,
                       keys=['nifty50', 'niftynext50'])
# indexing the hierarchical columns
nifty_2019['nifty50']['open']
# accessing a particular row
nifty_2019['nifty50'].loc['31 Dec 2019']
```

Summary

- Dataframe is a two-dimensional labeled data structure with columns of that can be of different types. In general it can be said to consist three components : data, index and columns. it is created using a call to the function DataFrame() which is overloaded.
- Dataframe can be created using numpy arrays, multiple series objects. Also, loc and iloc are used for indexing.

- Pandas provides many optimized functions to perform perfunctory data cleaning tasks (eg. dropping null values using df.dropna()).
- Functions can be applied to dataframe using df.apply().
- The template split-apply-aggregate is used to manipulate groups of data. A filter-split-apply-aggregate can be used to filter using a condition and then do grouping operations.

MCQ

1. Which of the following can be used to make a DataFrame?
    1. Series
    2. DataFrame
    3. Structured ndarray
    4. **All of the above**
2. A DataFrame can be sorted by
    1. by label
    2. by actual value
    3. **both a and b**
    4. neither
3. To sort a DataFrame by label
    1. sort_values() is used
    2. **sort_index() is used**
    3. both can be used
    4. neither can be used
4. Pandas index values must be
    1. unique
    2. hashable
    3. dict keys
    4. **all of the above**
5. What is the output of the code

```
s = pd.Series(np.random.randn(4))
print s.ndim
```

    1. 0
    2. **1**
    3. 2
    4. 3

6. from a dataset with student names and subject marks,

```
print results.groupby(level=0).agg(maximum)
```

to get maximum value for each student for respective tests, the maximum function can be defined as

```
1. def maximum(x):
   return np.max(x, axis=1)

2. def maximum(x):
   return np.max(x, axis=0)
```

```
3. def maximum(x):
   return max(x)

4. def maximum(x):
   return [y.max() for y in x]
```